Tema 7. El sistema de clases

7.1 Funciones Sobrecargadas

✓ Tienen sentido para algunos tipos, pero no todos

✓ Pueden tener definiciones distintas para cada tipo

Ejemplo: Consideremos los tipos

$$
\begin{array}{lcl}
\textbf{type } \mathit{Lado} & = & \mathit{Float} \\
\textbf{type } \mathit{Radio} & = & \mathit{Float} \\
\textbf{type } \mathit{Área} & = & \mathit{Float} \\
\textbf{data } \mathit{Cuadrado} & = & \mathit{UnCuadrado\ Lado\ } \textbf{deriving } \mathit{Show} \\
\textbf{data } \mathit{Círculo} & = & \mathit{UnCírculo\ Radio\ } \textbf{deriving } \mathit{Show}
\end{array}
$$

Tiene sentido definir una función para calcular el área de un $\mathit{Cuadrado}$:

$$
\begin{array}{lcl}
\mathit{área} & :: & \mathit{Cuadrado} \rightarrow \mathit{Área} \\
\mathit{área}\ (\mathit{UnCuadrado}\ l) & = & l * l
\end{array}
$$

O para un $\mathit{Círculo}$:

$$
\begin{array}{lcl}
\mathit{área} & :: & \mathit{Círculo} \rightarrow \mathit{Área} \\
\mathit{área}\ (\mathit{UnCírculo}\ r) & = & pi * r \wedge 2
\end{array}
$$

¿ Es el tipo de $\mathit{área} :: a \rightarrow \mathit{Área}$?

NO, no tiene sentido, p. ej., calcular el área de un Bool

$\mathit{área}$ tiene sentido para los tipos $\mathit{Cuadrado}$ y $\mathit{Círculo}$, pero no para Bool, luego no es polimórfica

Clases e Instancias

✓ $área$ tiene sentido para varios tipos pero *NO* para todos

✓ $área$ tiene una definición *DISTINTA* para cada tipo

$área$ es un ejemplo de función *Sobrecargada*

En Haskell, para definir función sobrecargada hay que crear una *clase* (conjunto de tipos que implementan la función):

```
class TieneÁrea t where
  área :: t → Área
```

✓ $TieneÁrea$ es el nombre de la clase (empieza por mayúscula)

✓ t es una variable de tipo que representa los tipos de la clase

✓ El *método* $área$ solo estará definido para los t que pertenezcan a la clase

Para incluir un tipo en una clase se realiza una *instancia*

```
instance TieneÁrea Cuadrado where
  área (UnCuadrado l) = l * l

instance TieneÁrea Círculo where
  área (UnCírculo r) = pi * r ^ 2
```

Uso:

```
? área (UnCuadrado 3)        -- Se usa primera instancia
9.0 :: Área

? área (UnCírculo 3)         -- Se usa segunda instancia
28.2743 :: Área

? área True                  -- No existe instancia adecuada
ERROR ...
```

7.2 Algunas Clases e Instancias predefinidas

✓ Haskell organiza los tipos predefinidos en clases de tipos.

Clase: conjunto de tipos para los que tiene sentido una serie de operaciones sobrecargadas.

✓ Algunas de las clases predefinidas:

- ◇ Eq tipos que definen igualdad: $(==)$ y $(/=)$
- ◇ Ord tipos que definen un orden: $(<=)$, $(<)$, $(>=)$, ...
- ◇ Num tipos numéricos: $(+)$, $(-)$, $(*)$, ...

Instancias: conjunto de tipos pertenecientes a una clase.

✓ Algunas instancias predefinidas:

- ◇ Eq tipos que definen igualdad: $Bool$, $Char$, Int, $Integer$, $Float$, $Double$, ...
- ◇ Ord tipos que definen un orden: $Bool$, $Char$, Int, $Integer$, $Float$, $Double$, ...
- ◇ Num tipos numéricos: Int, $Integer$, $Float$ y $Double$

La clase Eq

✓ Tipos *igualables*

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool

  -- Mínimo a implementar: (==) o bien (/=)

  x == y = not (x /= y)
  x /= y = not (x == y)
```

✓ Las definiciones en la clase de (==) y (/=) son *Métodos por defecto*

✓ Se usan si no se definen en las instancias

✓ Basta con definir uno de los dos

La clase Ord

✓ Tipos *ordenables*

```
data Ordering = LT | EQ | GT

class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
  compare              :: a -> a -> Ordering

  -- Mínimo a implementar: (<=) o bien compare

  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT

  x <= y = compare x y /= GT
  x <  y = compare x y == LT
  x >= y = compare x y /= LT
  x >  y = compare x y == GT

  max x y
    | x >= y    = x
    | otherwise = y

  min x y
    | x <= y    = x
    | otherwise = y
```

✓ *Ord* es *subclase* de *Eq*

✓ Cualquier tipo instancia de *Ord* debe ser instancia de *Eq* (no necesariamente lo contrario)

Las clases Show y Read

✓ Tipos mostrables

```
class Show a where
  show :: a → String
  ...
```

✓ Tipos leíbles

```
class Read a where
  ...

read   :: Read a ⇒ String → a
read s = ...
```

Las clases Num y Fractional

```
class (Eq a, Show a) ⇒ Num a where
  (+), (−), (∗) :: a → a → a
  negate        :: a → a
  abs, signum   :: a → a
  fromInteger   :: Integer → a

  -- Mínimo a implementar: todos, excepto negate o (−)

  x − y    = x + negate y
  negate x = 0 − x

class (Num a) ⇒ Fractional a where
  (/)    :: a → a → a
  recip  :: a → a
  ...

  recip x = 1 / x
  x / y   = x ∗ recip y
  ...
```

7.3 Instancias paramétricas

✓ Permiten definir un conjunto de instancias estableciendo condiciones

EJEMPLO: Si un tipo a dispone de la igualdad, es deseable que el tipo $[a]$ disponga también de igualdad (para cualquier a):

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \\
\quad [\,] \quad == [\,] \quad = \ True \\
\quad (x : xs) == (y : ys) = \ x == y\ \&\&\ xs == ys \\
\quad \_ \quad == \_ \quad = \ False
\end{array}
$$

✓ Una única declaración genera varias instancias: $[Int]$, $[Integer]$, $[Float]$, $[Double]$, $[Char]$, ...

✓ No todas las listas son instancias de Eq, solo aquellas cuyos elementos son instancias de Eq

7.4 Derivación de instancias

✓ La claúsula **deriving** permite generar instancias de ciertas clases predefinidas de forma automática.

✓ Aparece al final de una declaración de tipo

```
data Color = Rojo | Amarillo | Azul | Verde deriving (Eq, Ord, Show, Read)

? Rojo == Verde
False :: Bool

? Rojo < Verde
True :: Bool

? show Rojo
"Rojo" :: String

? read "Rojo" :: Color
Rojo :: Color
```

✓ Al derivar la clase Eq se usa *igualdad estructural*:

- ◇ Dos valores son iguales si tienen la misma forma

EJEMPLO:

```
infix 9 :/
data Racional = Integer :/ Integer deriving Eq
```

genera

```
instance Eq Racional where
  x :/ y == x' :/ y' = (x == x') && (y == y')

? 1 :/ 2 == 2 :/ 4
False :: Bool
```

✓ La igualdad estructural no es adecuada en este caso

Derivación de instancias (2)

EJEMPLO:

```
data Nat = Cero | Suc Nat deriving Eq
```

genera

```
instance Eq Nat where
  Cero  == Cero  = True
  Suc x == Suc y = (x == y)
  _     == _     = False
```

✓ La igualdad estructural es adecuada en este caso

Al derivar la clase Ord se usa *orden estructural*:

- ◇ Un dato es menor que otro si su constructor de datos aparece más a la izquierda en la definición de tipo
- ◇ Para dos datos con el mismo constructor, se comparan sus argumentos de izquierda a derecha

EJEMPLO:

```
data Nat = Cero | Suc Nat deriving (Eq, Ord)
```

genera la siguiente instancia de orden

```
instance Ord Nat where
  Cero  <= _     = True
  Suc x <= Suc y = x <= y
  _     <= _     = False
```

✓ El orden estructural es adecuado en este caso

7.5 Tipos sobrecargados: Contextos

✓ Los métodos de una clase solo pueden ser usados con tipos instancia de dicha clase

✓ Ésto queda reflejado en el tipo de los métodos:

```
? :t (==)
(==) :: Eq a => a -> a -> Bool

? :t (+)
(+) :: Num a => a -> a -> a

? True + False
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : True + False
*** Type       : Num Bool => Bool
```

✓ El *contexto* establece una restricción sobre el polimorfismo de la variable.

✓ *Propagación de contextos*:

Si función polimórfica usa una sobrecargada se convierte en sobrecargada.

```
doble   :: Num a => a -> a
doble x = x + x

elem            :: Eq a => a -> [a] -> Bool
x `elem` []     = False
x `elem` (y:ys) = x == y || x `elem` ys

f               :: (Ord a, Num a) => a -> a -> a
f x y
  | x < y       = x
  | otherwise   = x + y
```

Objetivos del tema

El alumno debe:

- ✓ Comprender el concepto de sobrecarga y diferenciarlo del concepto de polimorfismo
- ✓ Saber definir clases e instancias para sobrecargar funciones
- ✓ Conocer las clases predefinidas expuestas
- ✓ Conocer el significado de una instancia paramétrica
- ✓ Conocer la posibilidad de derivar instancias.
- ✓ Conocer la igualdad y el orden estructural generado para instancias derivadas
- ✓ Saber si una instancia derivada es adecuada
- ✓ Entender los tipos sobrecargados (*contextos*)
- ✓ Saber cuál es el tipo de una función sobrecargada a partir de su definición